Types:
fun f (g,h) = g (h) + 2;
(‘a -> int) * ‘a -> int

fun x y z = x+y+z
int -> int -> int -> int

2. Define a grammar which generates all palindromes over the alphabet {a,b,c}. Note: a palindrome is any string which reads the same backward as forward. For example, the empty string, "a", "aa", " aba", and "abba" are palindromes, whereas "ab" , "abb" , and "abc" are not.

S = aSa I bSb I cSc I T
T = a I b I c I empty

BNF Grammars:
Binary:
N :: = 0 I 1 I 0N I 1N

Palindrome:
S ::= aSa I aBa
B ::= bB I b

Palindrome over alphabet:
S ::= aSa I bSb I cSc I T
T : : = a I b I c I a a I b b I c c I ε

Grammars:
Regular - single terminal, with no operations, also not ambiguous.

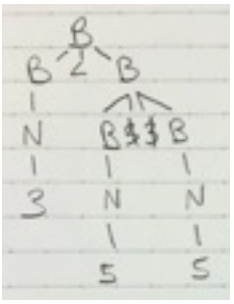
Context free - can have terminal symbols and variables can change

Context sensitive - terminal on left hand side.

terminal: +, -, /, (), etc

non-terminal: A or B, Type, etc

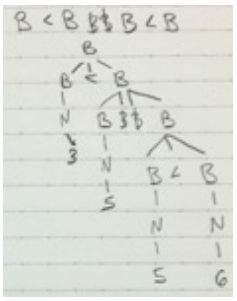
Grammar example: **B::= B < B I B && B I N**
Where N stands for all integer numbers.
— —
Give a parse tree for the expression 3 < 5 &&5:

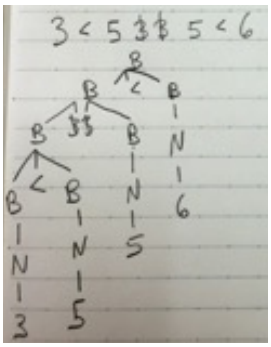


Dynamic Scoping
Variable is interpreted at run time, uses most recently defined call stack.

Static Scoping:
Variable is known at compile time.

Show its ambiguous:



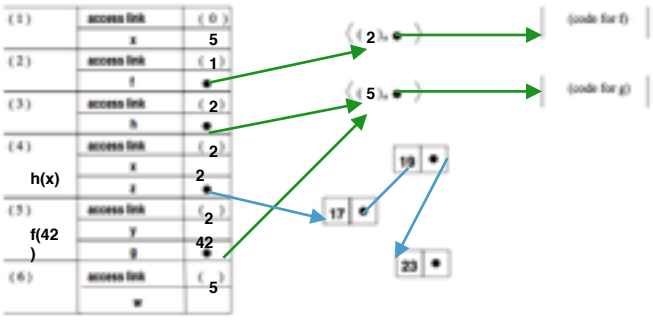


Consider the following ML code:

```
val x = 5;
fun f(y) = let fun g(w) = w+x+y in g end;
val h = let val x=2
            val z = [17, 19, 23]
        in f(42) end;
h(x);
```

– What is the value of h(x)?

h(5) = g(5)
y = 42
w = 5
x = 5
therefore h(x) = 52



ML Datatypes:
[1, true, 'aa"] would be defined as:
datatype MIX = A of int I B of bool I C of string

Person with name, age, dob:
datatype PERSON = firstname of string I lastname of string I Age of int I DOB of int

datatype PERSON = Person of string * string * int * string

Tree with polymorphic binary type:
datatype 'a tree = Leaves of 'a I NODE of 'a tree * 'a tree * 'a;

Shapes datatype:

First Class Functions:
- Can be declared in any scope
- passed as an argument to other functions
- returned as a result of functions

Higher Order Function Example:
val x = 4;
fun f(y) x * y
fun g(h) = let val x = 7 in h(x) + x
g(f);

ML Functions: ***@ == concat**

```
fun factorial 0 = 1
I    factorial n = n * factorial(n-1);
```

```
fun map(F,nil) = nil
I              map(F,x::xs) = F(x)::map(F,xs) ;
```

```
fun reduce f [x] = x
 I  reduce f (x :: y :: xs) = reduce f (f(x, y) :: xs);
```

Datatype:
datatype color = Red I Blue I Green;

ref v - creates a reference cell containing value v
! r -- returns the value contained in reference cell r

Pattern Matching:
fun factorial 0 = 1
I factorial n = n * factorial(n-1);

- fun merge (xs,nil) = xs
I merge (nil,ys) = ys
I merge (x::xs,y::ys) = if x<y then x :: merge(xs,y::ys) else y :: merge(x::xs,ys);
*val merge = fn : int list * int list -> int list*

- fun insert (x, nil) = [x]
= I insert (x,y::ys) = if x<y then x::y::ys else y :: insert (x,ys);

---

add_list : int list -> int
that adds numbers up in a list:

```
fun add_list nil = 0
I add_list[h::t]= ht sum_list[t];
```

```
fun sum_list(List) =
    let
            fun f(x:int, y:int) = x + y
    in
            G f List 0
    end;

fun sum_list(List) = G(fn(x, y) => x + y) List 0;
```

— - — — —

```
fun max_list(list) =
    let
            fun f(x:int, y:int) = if x > y then x else
y;
    in
            G f list 0
    end;

fun max_list(list) = G(fn (x, y) => if x > y then x
                          else y) list 0;
```

```
begin
 integer z ;

 procedure p (x);
  integer x; //type of the formal parameter
    begin
        x := x+1;
        z := z+2
    end
                Pass by Value - 3

  z := 1;       Pass by Reference - 4
  p(z);
  print z       Pass by Value/ Result - 2
end
```

```
— fun name(BS(n)) = n
  | name(MS(n,s)) = n
  | name(PhD(n,f)) = n;
val name = fn : student → name
```

— datatype student = BS of name | MS of name*school | PhD of name*faculty;

Lambda Calculus:

Substitution - $(\lambda x.M)N = [N/x]M,$

$(\lambda f \lambda x.f(fx))(\lambda y.y+1)2$
$\rightarrow (\lambda x.(\lambda y.y+1)((\lambda y.y+1)x))2$
$\rightarrow (\lambda x.(\lambda y.y+1)(x+1))2$
$\rightarrow (\lambda x.(x+1+1))2$
$\rightarrow (2+1+1).$

- Renaming - Alpha axiom
- apply a function - Beta axiom

Curried Functions:
curried: ((‘a * ‘b) -> ‘c) -> (‘a -> (‘b -> ‘c))
uncurried: (‘a -> (‘b -> ‘c)) -> ((‘a * ‘b) -> ‘c)
—
fun curry f x y = f (x, y);
fun uncurry f (x, y) = f x y;

Scope:
a region of text in which a declaration is visible.

Lifetime:
the duration, during a run of a program, during which a location is allocated as the result of a specific declaration.

Activation Records:
- **Control Link**: Pointer to top of previous AR
- Activation record made when entering new block
- **access link** of an activation record points to the activation record of the closest enclosing block in the program

ML tail recursive compute length of list:

```
fun length acc 0 = acc
I    length acc(x::y) = length((acc+1), y)
```

This will pass an accumulator value as well as the list. If the list is empty, acc will return nil. However if the list is not it will trace through the list adding 1 to acc while also removing the head of the list.

Parameter Passing:
Pass by Value - making a copy in memory of the actual parameters value that is passed in, a copy of the contents of the actual parameter.

Pass by Reference - (pass by address), a copy of the address of the actual parameter is stored.

Pass by Value/ Result - reference is passed in but the value of the reference doesn't change until you exit the current scope

Call By:

```
begin                          Call by Value : i =2
    integer i;
    procedure pass(x, y);
    integer x, y;              Call by Reference: i = 4
    x : x + 1;                 Call by Result: i = 3
    y : x + 1;
    x : y + 1;
    i : i + 1;
end
i : = 1;
pass (i, i);
print i
end
```

Key Notes:
- ( ) : unit = type void
- Let…in…end <=> {…;….}
- Big substitutions: when subbing change bounded variables then sub.